Виртуальные функции и полиморфизм.

Виртуальная функция — это функция, объявленная с ключевым словом virtual в базовом классе и переопределенная в одном или в нескольких производных классах. Виртуальные функции являются особыми функциями, потому что при вызове объекта производного класса с помощью указателя или ссылки на него C++ определяет во время исполнения программы, какую функцию вызвать, основываясь на типе объекта. Для разных объектов вызываются разные версии одной и той же виртуальной функции. Класс, содержащий одну или более виртуальных функций, называется полиморфным классом (polymorphic class).

Виртуальная функция объявляется в базовом классе с использованием ключевого слова virtual. Когда же она переопределяется в производном классе, повторять ключевое слово virtual нет необходимости, хотя и в случае его повторного использования ошибки не возникнет.

В качестве первого примера виртуальной функции рассмотрим следующую короткую программу:

```
// небольшой пример использования виртуальных функций
#include <iostream.h>
class Base {
public:
virtual void who() { // определение виртуальной функции
cout << *Base\n";
}
};
class first_d: public Base {
public:
void who() { // определение who() применительно к first_d
cout << "First derivation\n";
}
};
class seconded: public Base {
public:
void who() { // определение who() применительно к second_d
cout << "Second derivation\n*";
}
};
int main()
{
Base base_obj;
Base *p;
first_d first_obj;
second_d second_obj;
p = &base_obj;
p->who(); // доступ к who класса Base
p = &first_obj;
p->who(); // доступ к who класса first_d
p = &second_ob;
p->who(); // доступ к who класса second_d
return 0;
}
```

Программа выдаст следующий результат:

```
Base
First derivation
Second derivation
```

Проанализируем подробно эту программу, чтобы понять, как она работает.

Как можно видеть, в объекте Base функция who() объявлена как виртуальная. Это означает, что эта функция может быть переопределена в производных классах. В каждом из классов first_d и second_d функция who() переопределена. В функции main() определены три переменные. Первой является объект base_obj, имеющий тип Base. После этого объявлен указатель p на класс Base, затем объекты first_obj и second_obj, относящиеся к двум производным классам. Далее указателю p присвоен адрес объекта base_obji вызвана функция who(). Поскольку эта функция объявлена как виртуальная, то C++ определяет на этапе исполнения, какую из версий функции who() употребить, в зависимости от того, на какой объект указывает указатель p. В данном случае им является объект типа Base, поэтому исполняется версия функции who(), объявленная в классе Base. Затем указателю p присвоен адрес объекта first_obj. (Как известно, указатель на базовый класс может быть использован для любого производного класса.) После того, как функция who() была вызвана, C++ снова анализирует тип объекта, на который указывает p, для того, чтобы определить версию функции who(), которую необходимо вызвать. Поскольку p указывает на объект типа first_d, то используется соответствующая версия функции who(). Аналогично, когда указателю p присвоен адрес объекта second_obj, то используется версия функции who(), объявленная в классе second_d.

Наиболее распространенным способом вызова виртуальной функции служит использование параметра функции. Например, рассмотрим следующую модификацию предыдущей программы:

```
/* Здесь ссылка на базовый класс используется для доступа к виртуальной функции */
#include <iostream.h>
class Base {
public:
virtual void who() { // определение виртуальной функции
cout << "Base\n";
}
};
class first_d: public Base {
public:
void who () { // определение who() применительно к first_d
cout << "First derivation\n";
}
};
class second_d: public Base {
public:
void who() { // определение who() применительно к second_d
cout << "Second derivation\n*";
}
};
// использование в качестве параметра ссылки на базовый класс
void show_who (Base &r) {
r.who();
}
```

```
int main()
{
Base base_obj;
first_d first_obj;
second_d second_obj;
show_who (base_ob j) ; // доступ к who класса Base
show_who(first_obj); // доступ к who класса first_d
show_who(second_obj); // доступ к who класса second_d
return 0;
}
```

Эта программа выводит на экран те же самые данные, что и предыдущая версия. В данном примере функция show_who() имеет параметр типа ссылки на класс Base. В функции main() вызов виртуальной функции осуществляется с использованием объектов типа Base, first_d и second_d. Вызываемая версия функции who() в функции show_who() определяется типом объекта, на который ссылается параметр при вызове функции.

Ключевым моментом в использовании виртуальной функции для обеспечения полиморфизма времени исполнения служит то, что используется указатель именно на базовый класс. Полиморфизм времени исполнения достигается только при вызове виртуальной функции с использованием указателя или ссылки на базовый класс. Однако ничто не мешает вызывать виртуальные функции, как и любые другие «нормальные» функции, однако достичь полиморфизма времени исполнения на этом пути не удается.

На первый взгляд переопределение виртуальной функции в производном классе выглядит как специальная форма перегрузки функции. Но это не так, и термин перегрузка функции не применим к переопределению виртуальной функции, поскольку между ними имеются существенные различия. Во-первых, функция должна соответствовать прототипу. Как известно, при перегрузке обычной функции число и тип параметров должны быть различными. Однако при переопределении виртуальной функции интерфейс функции должен в точности соответствовать прототипу. Если же такого соответствия нет, то такая функция просто рассматривается как перегруженная и она утрачивает свои виртуальные свойства. Кроме того, если отличается только тип возвращаемого значения, то выдается сообщение об ошибке. (Функции, отличающиеся только типом возвращаемого значения, порождают неопределенность.) Другим ограничением является то, что виртуальная функция должна быть членом, а не другом класса, для которого она определена. Тем не менее виртуальная функция может быть другом другого класса. Хотя деструктор может быть виртуальным, но конструктор виртуальным быть не может.

В силу различий между перегрузкой обычных функций и переопределением виртуальных функций будем использовать для последних термин переопределение (overriding).

Если функция была объявлена как виртуальная, то она и остается таковой вне зависимости от количества уровней в иерархии классов, через которые она прошла. Например, если класс second_d получен из класса first_d, а не из класса Base, то функция who() останется виртуальной и будет вызываться корректная ее версия, как показано в следующем примере:

```
// порождение от first_d, а не от Base
class second_d: public first_d {
public:
void who() { // определение who() применительно к second_d
cout << "Second derivation\n*";
```

```
}
};
```

Если в производном классе виртуальная функция не переопределяется, то тогда используется ее версия из базового класса. Например, запустим следующую версию предыдущей программы:

```
#include <iostream.h>
class Base {
public:
virtual void who() {
cout << "Base\n";
}
};
class first_d: public Base {
public:
void who() {
cout << "First derivation\n";
}
};
class second_d: public Base {
// who() не определяется
};
int main()
{
Base base_obj;
Base *p;
first_d first_obj; ,
second_d second_obj;
p = &base_obj;
p->who(); // доступ к who класса Base
p = &first obj;
p->who(); // доступ к who класса first_d
p = &sepond_ob;
p->who(); /* доступ к who() класса Base, поскольку second_d не переопределяет */
return 0;
}
```

Эта программа выдаст следующий результат:

```
Base
First derivation
Base
```

Надо иметь в виду, что характеристики наследования носят иерархический характер. Чтобы проиллюстрировать это, предположим, что в предыдущем примере класс second_d порожден от класса first_d вместо класса Base. Когда функцию who() вызывают, используя указатель на объект типа second_d (в котором функция who() не определялась), то будет вызвана версия функции who(), объявленная в классе first_d, поскольку этот класс — ближайший к классу second_d. В общем случае, когда класс не переопределяет виртуальную функцию, C++ использует первое из определений, которое он находит, идя от потомков к предкам.

виртуальные функции в комбинации с производными типами позволяют языку C++ поддерживать полиморфизм времени исполнения. Этот полиморфизм важен для объектно-ориентированного программирования, поскольку он позволяет переопределять функции базового класса в классах-потомках с тем, чтобы иметь их версию применительно к данному конкретному классу. Таким образом, базовый класс определяет общий интерфейс, который имеют все производные от него классы, и вместе с тем полиморфизм позволяет производным классам иметь свои собственные реализации методов. Благодаря этому полиморфизм часто определяют фразой «один интерфейс — множество методов».

Успешное применение полиморфизма связано с пониманием того, что базовые и производные классы образуют иерархию, в которой переход от базового к производному классу отвечает переходу от большей к меньшей общности. Поэтому при корректном использовании базовый класс обеспечивает все элементы, которые производные классы могут непосредственно использовать, плюс набор функций, которые производные классы должны реализовать путем их переопределения.

Наличие общего интерфейса и его множественной реализации является важным постольку, поскольку помогает программистам разрабатывать сложные программы. Например, доступ ко всем объектам, производным некоторого базового класса, осуществляется одинаковым способом, даже если реальные действия этих объектов отличаются при переходе от одного производного класса к другому. Это означает, что необходимо запомнить только один интерфейс, а не несколько. Более того, отделение интерфейса от реализации позволяет создавать библиотеки классов, поставляемые независимыми разработчиками. Если эти библиотеки реализованы корректно, то они обеспечивают общий интерфейс, и их можно использовать для вывода своих собственных специфических классов.

Чтобы понять всю мощь идеи «один интерфейс — множество методов», рассмотрим следующую короткую программу. Она создает базовый класс figure. Этот класс используется для хранения размеров различных двумерных объектов и для вычисления их площадей. Функция set_dim() является стандартной функцией-членом, поскольку ее действия являются общими для всех производных классов. Однако функция show_area() объявляется как виртуальная функция, поскольку способ вычисления площади каждого объекта является специфическим. Программа использует класс figure для вывода двух специфических классов square и triangle.

```
#include <iostream.h>
class figure {
protected:
double x, y;
public:
void set_dim( double i, double j) {
x = i;
y = j;
}
virtual void show_area() {
cout << "No area computation defined ";
cout << "for this class. \n";
}
};
class triangle: public figure {
public:
void show_area() {
```

```
cout << "Triangle with height ";
cout << x << " and base " << y;
cout << " has an area of ";
cout << x * 0.5 * y << ". \n";
}
};
class square: public figure {
public:
void show_area() {
cout << "Square with dimensions ";
cout << x << "x" << y;
cout << " has an area of ";
cout << x * y << ". \n";
}
};
int main ( )
{
figure *p; /* создание указателя базового типа */
triangle t; /* создание объектов порожденных типов */
square s;
p = &t;
p->set_dim(10.0, 5.0);
p->show_area();
p = &s;
p->set_dim(10.0, 5.0);
p->show_area ();
return 0;
}
```

Как можно видеть на основе анализа этой программы, интерфейс классов square и triangle является одинаковым, хотя оба обеспечивают свои собственные методы для вычисления площади каждой из фигур. На основе объявления класса figure можно вывести класс circle, вычисляющий площадь, ограниченную окружностью заданного радиуса. Для этого необходимо создать новый производный класс, в котором реализовано вычисление площади круга. Вся сила виртуальной функции основана на том факте, что можно легко вывести новый класс, разделяющий один и тот же общий интерфейс с другими подобными объектами. В качестве примера здесь показан один из способов реализации:

```
class circle: public figure {
public:
void show_area() {
cout << "Circle with radius ";
cout << x;
cout << "has an area of ";
cout << 3.14 * x * x;
}
};
```

Прежде чем использовать класс circle, посмотрим внимательно на определение функции show_area(). Обратим внимание, что она использует только величину x, которая выражает радиус. Как известно, площадь круга вычисляется по формуле $\pi R2$. Однако функция set_dim(), определенная в классе figure, требует не одного, а двух аргументов.

Поскольку класс circle не нуждается во второй величине, то как же нам быть в данной ситуации?

Имеются два пути для решения этой проблемы. Первый заключается в том, чтобы вызвать set_dim(), используя в качестве второго параметра фиктивный параметр, который не будет использован. Недостатком такого подхода служит необходимость запомнить этот исключительный случай, что по существу нарушает принцип «один интерфейс — множество методов».

Лучшее решение данной проблемы связано с использованием параметра y в set_dim() со значением по умолчанию. В таком случае при вызове set_dim() для круга необходимо указать только радиус. При вызове set_dim() для треугольника или прямоугольника укажем обе величины. Ниже показана программа, реализующая этот подход:

```
#include <iostream.h>
class figure {
protected:
double x, y;
public:
void set_dim (double i, double j=0) {
x = i;
y = j;
}
virtual void show_area() {
cout << "No area computation defined ";
cout << "for this class .\n";
}
};
class triangle: public figure {
public:
void show_area() {
cout << "Triangle with height ";
cout << x << " and base " << y;
cout << " has an area of ";
cout << x * 0.5 * y << ". \n";
}
};
class square: public figure {
public:
void show_area() {
cout << "Square with dimensions ";
cout << x << "x" << y;
cout << " has an area of ";
cout << x * y << ". \n";
}
};
class circle: public figure {
public:
void show_area() {
cout << "Circle with radius ";
cout << x;
cout << has an area of ";
cout << 3.14 * x * x;
}
```

```
};
int main ( )
{
figure *p; /* создание указателя базового типа */
triangle t; /* создание объектов порожденных типов */
square s;
circle c;
p = &t;
p->set_dim(10.0, 5.0);
p->show_area ();
p = &s;
p->set_dim(10.0, 5.0);
p->show_area ();
p = &c;
p->set_dim(9. 0) ;
p->show_area ();
return 0;
}
```

Этот пример также показывает, что при определении базового класса важно проявлять максимально возможную гибкость. Не следует налагать на программу какие-то ненужные ограничения.